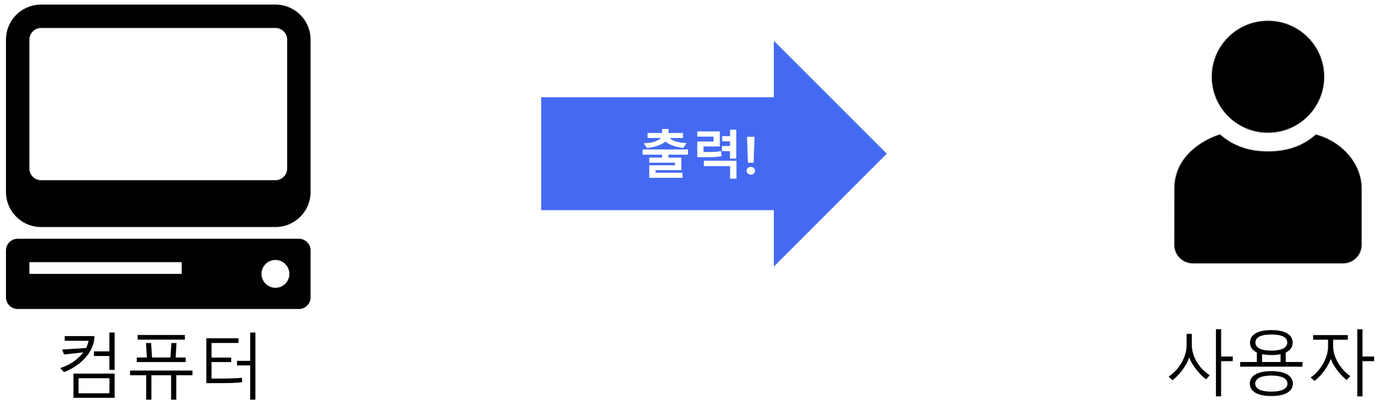
파이썬 기초

파이썬 기본 문법 학습

출력이란

- 우리가 원하는 정보나 자료를 컴퓨터가 출력하게 할 수 있습니다



print(“출력할 내용”)

```
print(“나의 꿈은 파이썬 정복!”)
```

실행 결과

```
나의 꿈은 파이썬 정복!
```

1. 숫자형

- **숫자**로 이루어진 자료형
- **정수**나 **실수**
- 숫자 끼리 **연산** 가능
- 정수(integer) = int
- 실수(float) = float

```
3 # 정수(integer)

3.14 # 실수(float)

3+4j # 복소수
```

2. 문자열 (String)

- 문자나 문자들을 늘어놓은 것
- **큰 따옴표(“”)**와 **작은 따옴표(‘’)**로 구분

```
‘Hello!’

‘3.14’ # 작은 따옴표
OK

“3.14” # 큰 따옴표 OK
```

문자나 문자들을 **늘어놓은 것**

3. 리스트

- **여러 자료를 순서대로 보관**하는 자료형.
- 다른 종류의 자료를 **함께** 담을 수 있음

```
[] # 빈 리스트
['a', 'b']
['a', 2] # 다른 자료형을 함께!
```

4. 논리 자료형(Boolean)

- **참(True)** 혹은 **거짓(False)**을 나타내는 자료형
- True 는 숫자 1, False는 숫자 0에 대응함

```
print(False)    # False
print(True)     # True
```

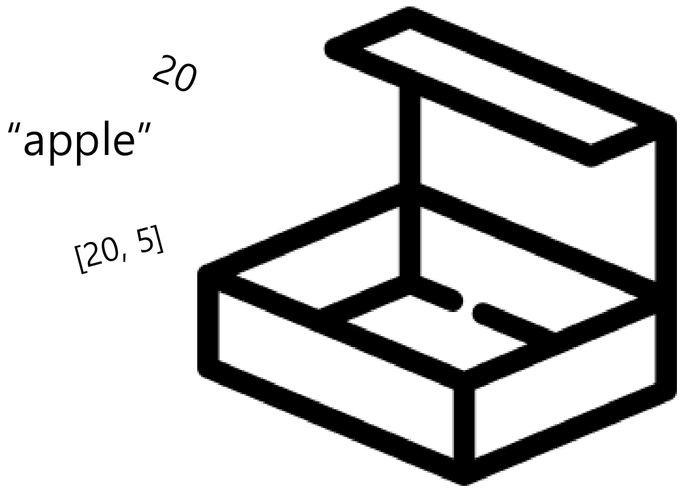
참고. 주석(comment)

- 주석은 컴퓨터가 **무시**
- 한 줄 주석은 #, 여러 줄 주석은 """ """ 혹은 ''' '''(따옴표 3번)

```
# 주석 처리한 말들은

"""""
컴퓨터가
실행하지
않아요!
"""""
```

변수란

- 자료를 '그릇'에 담아서 보관, 사용하면 편리함.
- 이를 **변수(variable)**라 함.



변수 사용방법

- 변수 이름 = 자료

```
num = 10 # 숫자
name = “Michael” # 문자열
grade = [‘A+’, ‘B+’, ‘A0’] # 리스트
```

변수 이름 = 자료

변수 이름짓는 방법

YES :

숫자, 알파벳, 한글, 언더바(_) 등을 사용할 수 있다

python = “파이썬”

파이썬 = “Python”

python_123 = 123

변수 이름짓는 방법

NO :

1. 변수이름이 **숫자로 시작**하면 안된다

```
1name = “Orange”
```

2. **숫자로만 구성**된 변수 이름 금지

```
1234 = 5678
```

변수 이름짓는 방법

NO :

3. 파이썬 문법에서 사용되는 **예약어**

   (예 : for, if ...) 사용 금지

```
print = “Orange”
print(print)
```

4. **공백 문자**( )와 **연산자**(+, -, %)

   사용 금지

```
fruit name = “Orange”
fruit+name = “Orange”
print(fruit name, fruit+name)
```

수학의 사칙연산

+	−	×	÷
더하기	빼기	곱하기	나누기

숫자형 자료의 사칙연산

+	-	*	/
더하기	빼기	곱하기	나누기

숫자형 자료의 사칙연산

```
print(3+5)      # 8
print(3-5)      # -2
print(3*5)      # 15
print(3/5)      # 0.6
```

숫자형 자료의 특수연산

//	%	**
몫 연산자	나머지 연산자	제곱 연산자

숫자형 자료의 특수연산

```
print(13//5)      # 2
print(13%5)       # 3
print(2**4)       # 16
```

문자형 자료의 연산

\+

이어 붙이기

with 문자열

*

반복하기

with 숫자

문자형 자료의 연산

```
print(“안녕” + “하세요”)
# 안녕하세요
print(“안녕” * 3)
# 안녕안녕안녕
```

질문!

‘rescue’와 ‘secure’은 **다른** 문자열일까요?

[1, 2, 3]과 [3, 2, 1]은 **다른** 리스트일까요?

질문!

‘rescue’와 ‘secure’은 **다른** 문자열일까요?

[1, 2, 3]과 [3, 2, 1]은 **다른** 리스트일까요?

Why?

원소의 배치 순서가 다르기 때문이죠!

인덱스

문자열과 리스트 자료형은 여러 원소로 이루어져 있고

각각의 **위치**를 **0**부터 순서대로 매길 수 있습니다

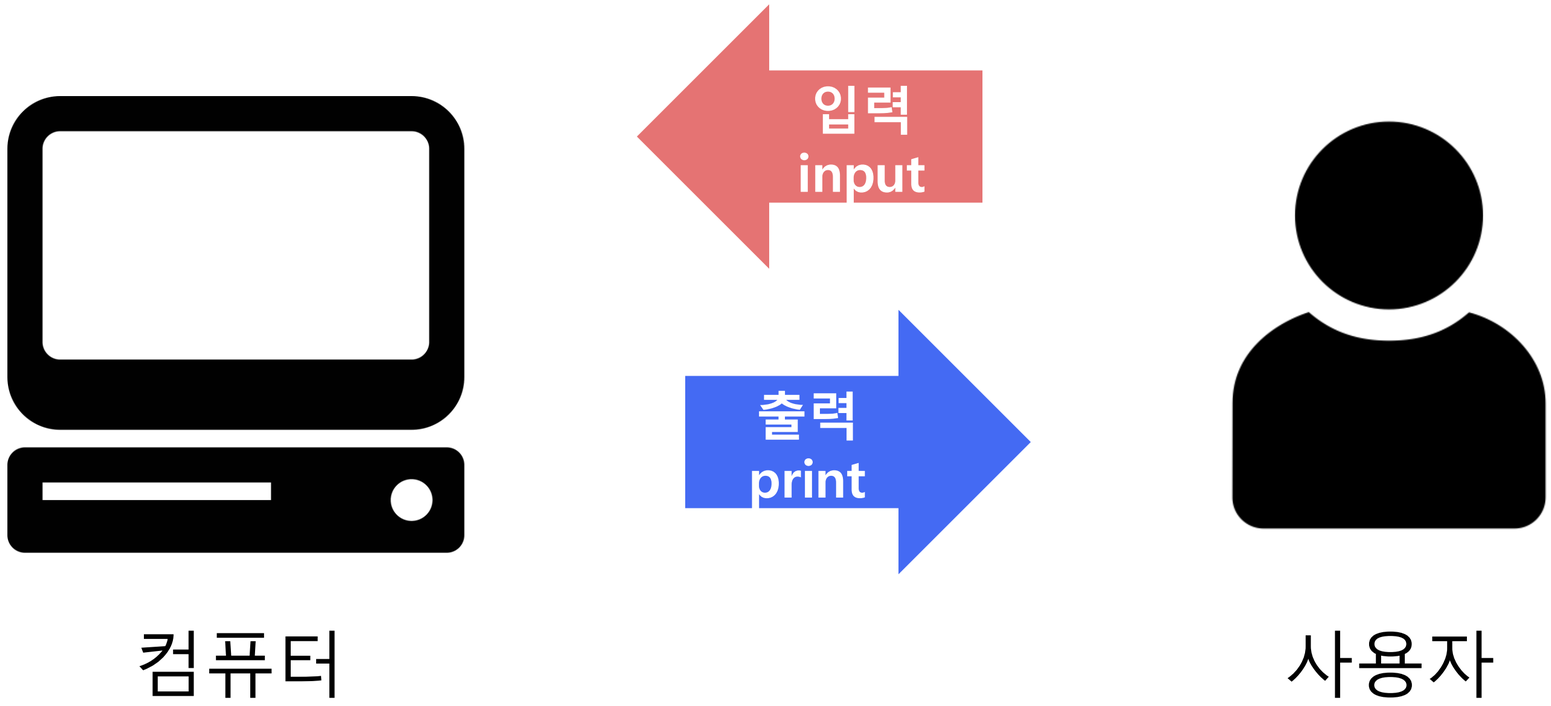
```
“R e a d y”
 0 1 2 3 4
```

```
[2, 4, 6, 8]
 0  1  2  3
```

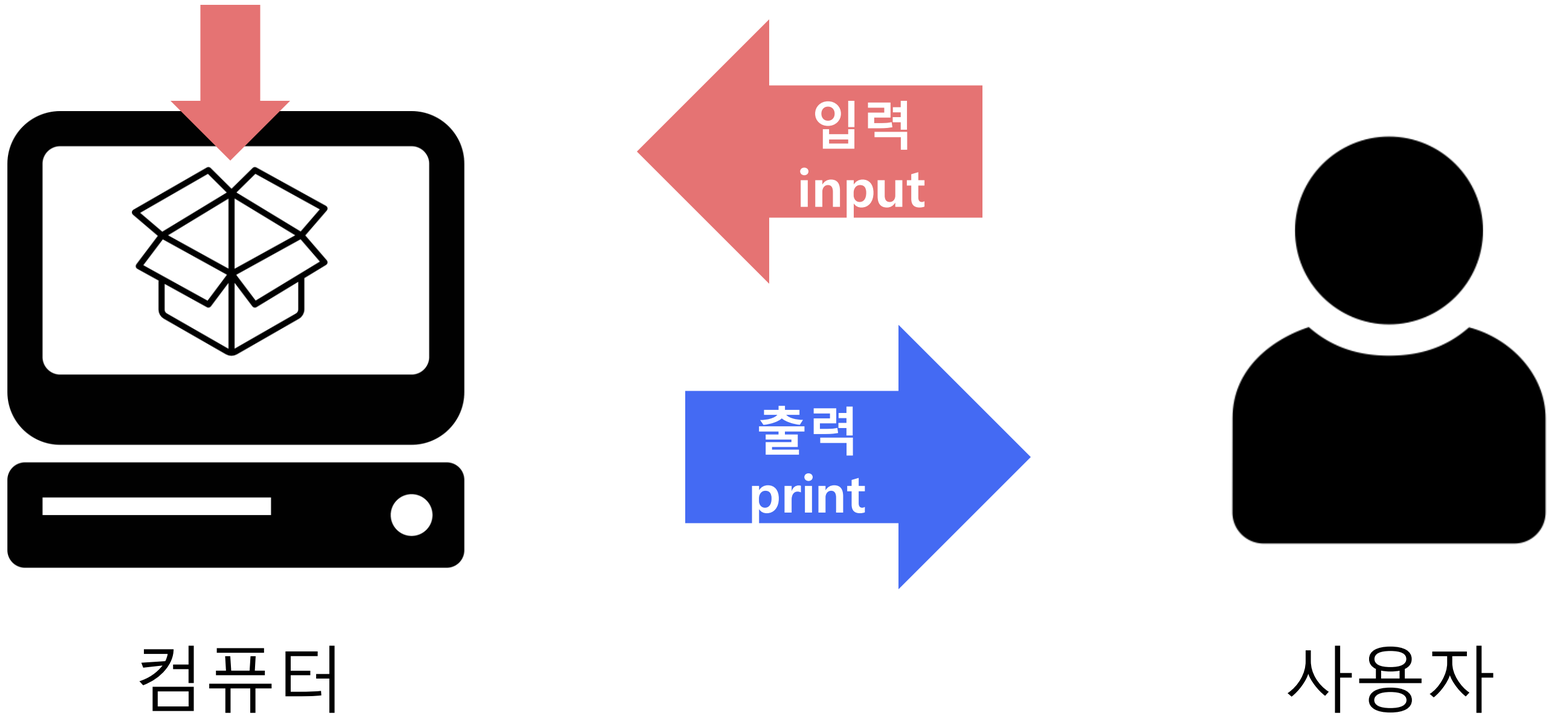
입력이란

- 우리가 원하는 정보나 자료를 컴퓨터에게 전달할 수 있음

입력 vs 출력



입력 vs 출력



입력

- 변수 = **input**()
- 터미널에 값을 “입력” + Enter
- 무엇을 입력하든 **“문자열”**로 입력됨

```
num = input()
>>> 터미널에 입력값을 넣어주세요...
```

```
num = input(“숫자를 입력하세요: ”)
print(num)
print(type(num))
```

```
실행결과

숫자를 입력하세요: 10
10
<class ‘str’>
```

형 변환

- 문자열을 숫자로 바꾸고 싶다면 자료형을 변환(형 변환) 사용

```
num1 = “5”
num2 = “100”
print(num1 + num2)
print(int(num1) + int(num2))
```

실행결과

```
5100
105
```

형 변환

구분	이름	예시	결과
정수	**int**eger	int(“1005”)	1005
실수	**float**	float(“3.14159”)	3.14159
문자열	**str**ing	str(3.14159)	“3.14159”
리스트	**list**	list(“1005”)	[“1”, “0”, “0”, “5”]

비교 연산자

- 숫자나 문자의 값을 **비교**하는 연산자
- 주어진 진술이 참이면 **True**, 거짓이면 **False**

```
print(3 < 5)      #True
print(7 == 5)     #False
print(2 >= 10)    #False
print(5 != 10)    #True
```

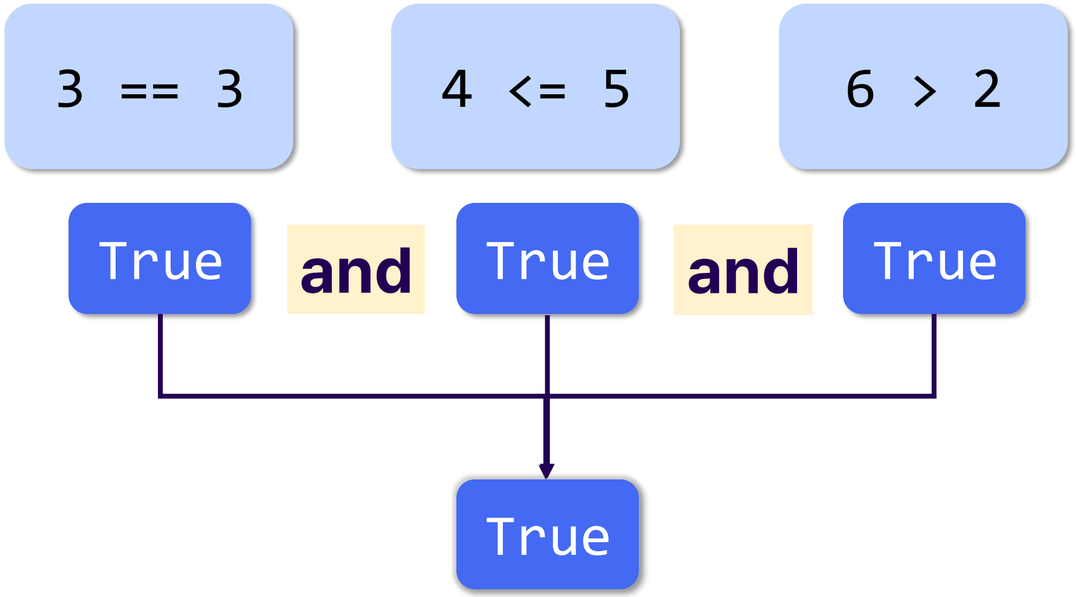
비교 연산자의 종류

==	같다 ('='는 변수 선언을 위해 사용하는 연산자)
!=	다르다
>	왼쪽이 더 크다
<	오른쪽이 더 크다
>=	왼쪽이 크거나 같다
<=	오른쪽이 크거나 같다

논리 자료형의 연산 - AND

- 각 논리가 **모두** True여야 True!

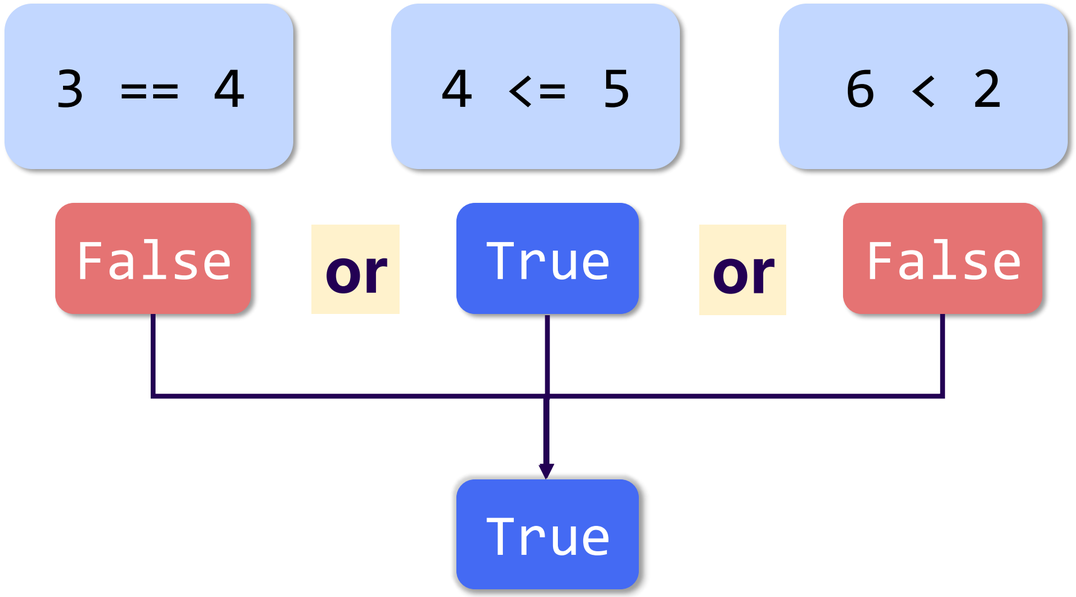
```
>>> print(3==3 and 4<=5 and 6>2)
#세 항이 모두 True이므로, True!
True
```



논리 자료형의 연산 - OR

- 논리들 중 True가 **하나라도 존재**하면 True!

```
>>> print(3==4 or 4<=5 or 6<2)
#세 항이 모두 True이므로, True!
True
```

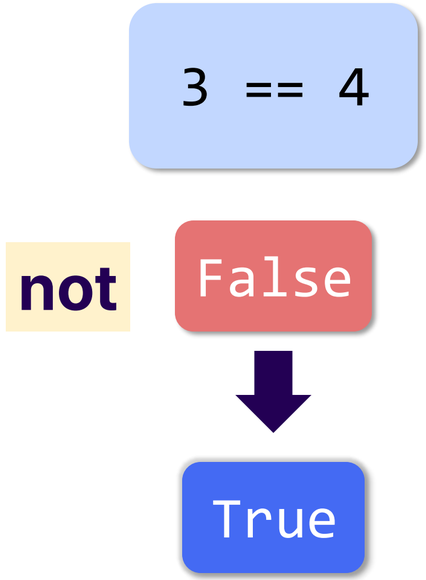


논리 자료형의 연산 - NOT

- 논리값을 **뒤집는다**!

True ⟷ False

```
>>> print(not 3==4)
# False에 Not을 붙였으므로, True!
True
```

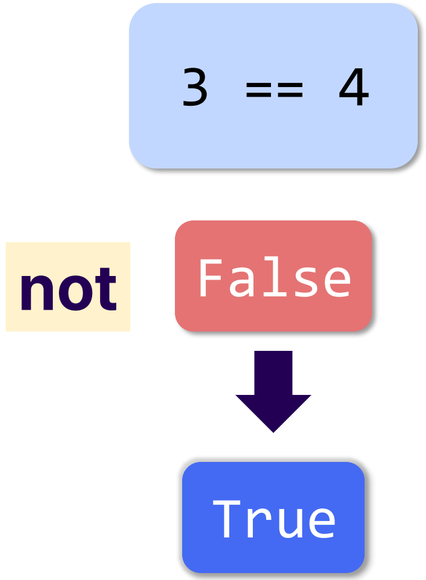


논리 자료형의 연산 - NOT

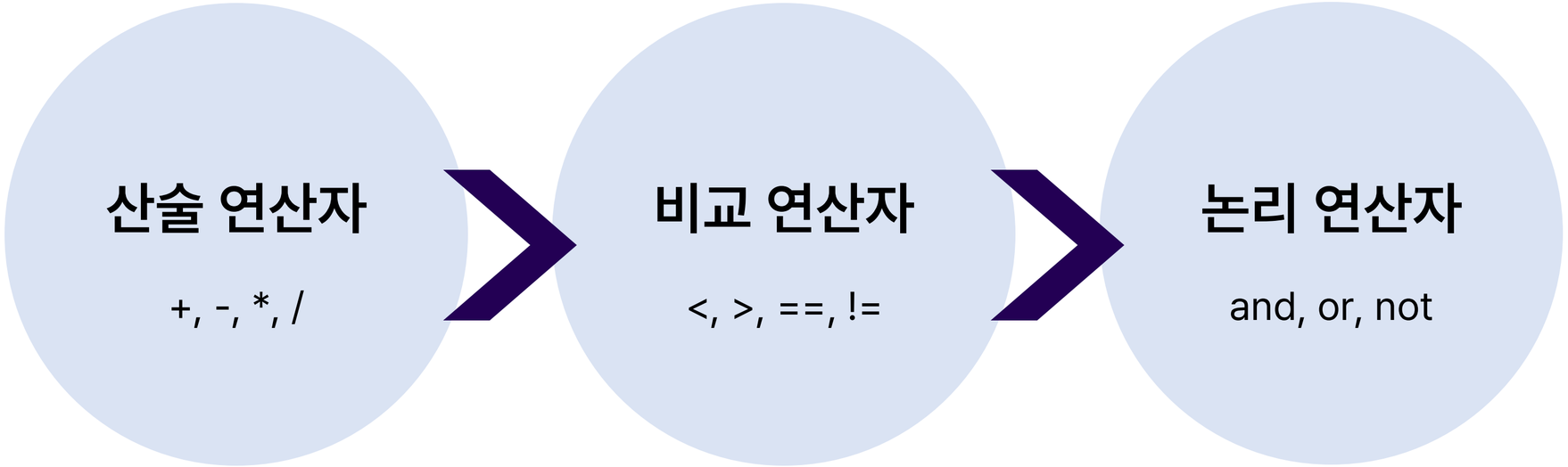
- 논리값을 **뒤집는다**!

True ⟷ False

```
>>> print(not 3==4)
# False에 Not을 붙였으므로, True!
True
```



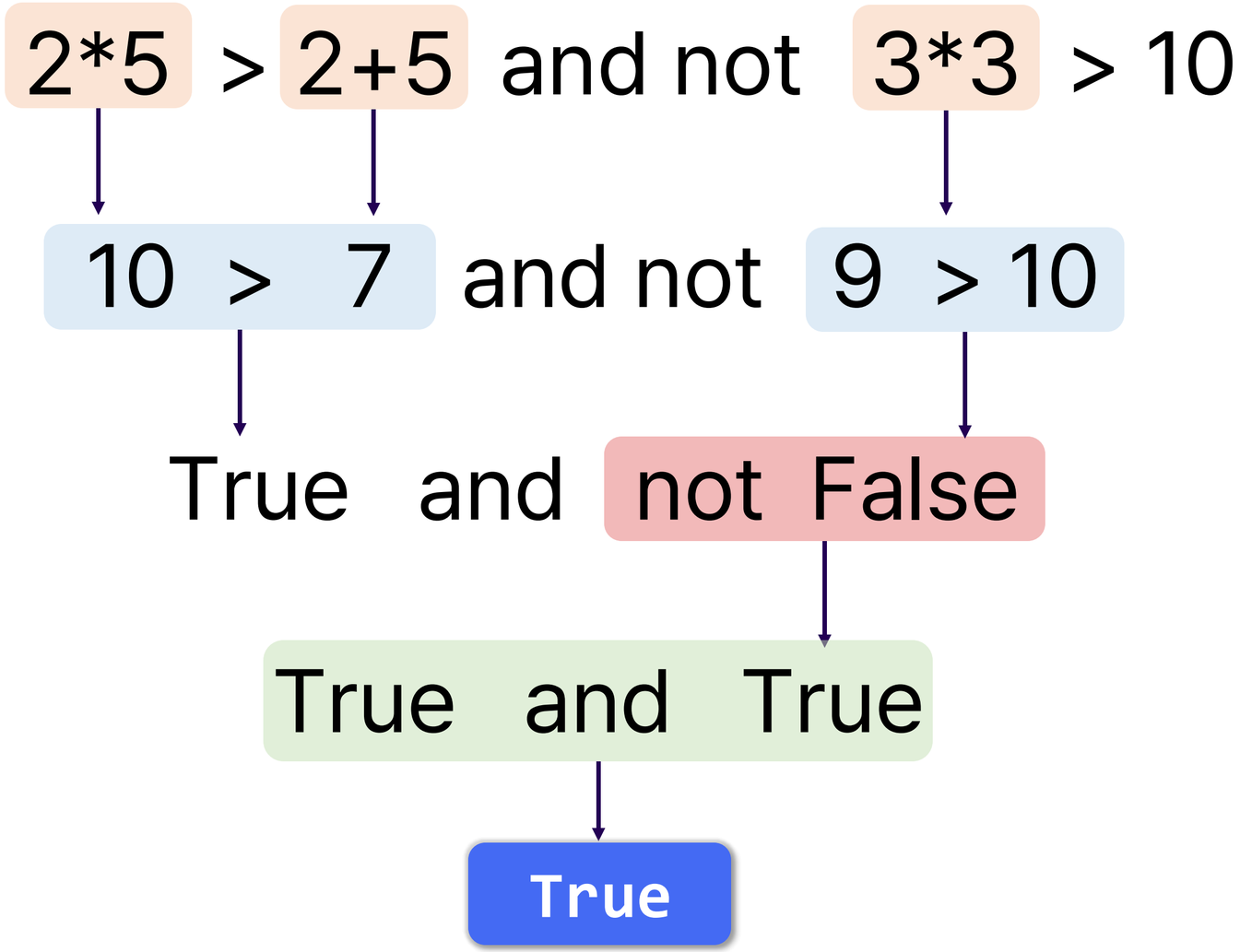
산술, 비교, 논리 연산자 우선순위



산술, 비교, 논리 연산자 우선순위

```
2*5 > 2+5 and not 3*3 > 10
```

산술, 비교, 논리 연산자 우선순위



파이썬의 조건문 – if 문

만약 상품A의 리뷰 개수 >0 이면, 제품명을 출력하라!

if 조건 명령

if문

- 조건이 **True**일 때, 명령 실행
- if문에 들어갈 명령들은 **같은 들여쓰기**로 구분

```
if 조건식:
....<수행할 명령>
....<수행할 명령>
....~~~~~~~
```

```
x = 10
if x > 5 :
....print(“x는 5보다 큽니다.”)
```

실행결과

```
x는 5보다 큽니다
```

if - else문

- if문에서 조건을 만족하지 못한다면

만약 상품 A의 리뷰 개수 > 0 이면, 제품명을 출력하라!

`if` 조건 명령

아니면 '리뷰 없음'을 출력해라!

`else`

if - else문

- if문에서 조건을 만족하지 못한다면

if 조건 : 리뷰 개수 > 0

else의 조건 : 리뷰 개수 <= 0

if - else문

- 조건이 True면 if문, False면 else문 실행

```
if 조건식:
....<수행할 명령>
else:
....<수행할 명령>
```

```
x = 15
if x < 10:
....print("x는 10보다 작습니다.")
else :
....print("x는 10보다 큽니다.")
```

실행결과

```
x는 10보다 큽니다.
```

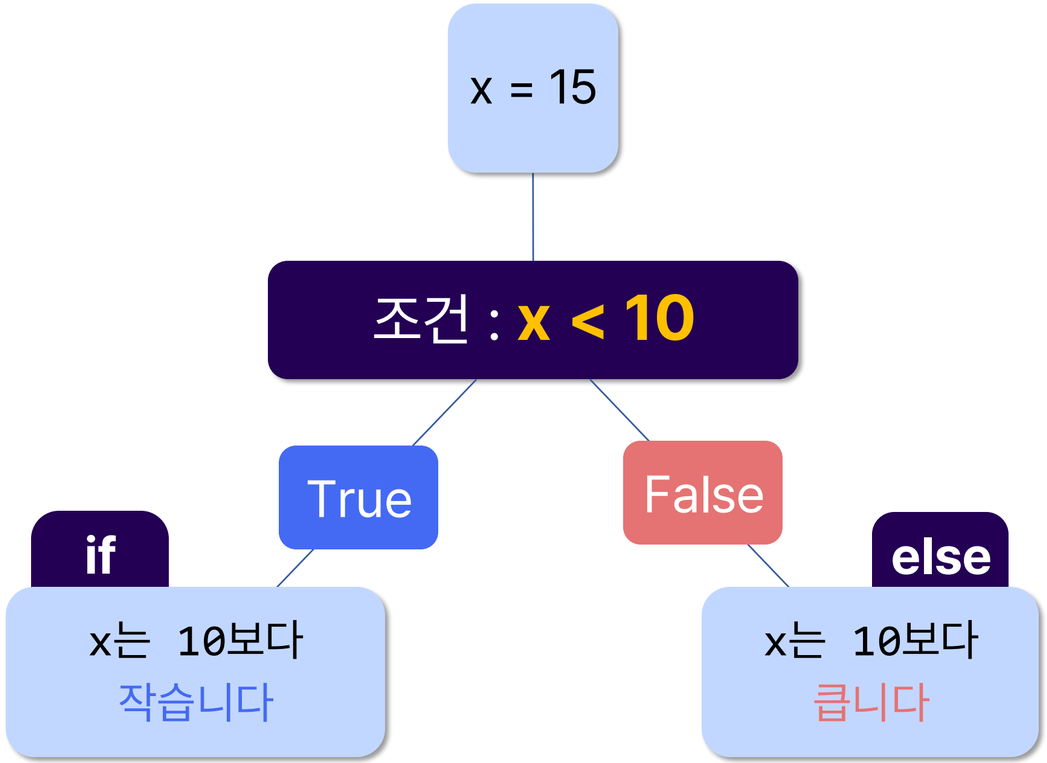
if - else문

- 조건이 True면 if문, False면 else문 실행

```
x = 15
if x < 10:
....print("x는 10보다 작습니다.")
else :
....print("x는 10보다 큽니다.")
```

실행결과

```
x는 10보다 큽니다.
```



if - elif문

- if문에서 조건을 만족하지 못한다면

만약 평점이 4.5 이상이면 '평점이 매우 높습니다'를 출력해라
if

만약 평점이 3.0 이상 4.5 미만이면 '평점이 준수합니다'를 출력해라
elif

만약 평점이 2.0이상 3.0미만이면 '평점이 낮아 개선이 필요합니다'를 출력해라
elif

if - elif문

- 조건 1이 **True**면 if문
- 조건 1이 **False**이면서 조건 2가 **True**면 elif문 실행

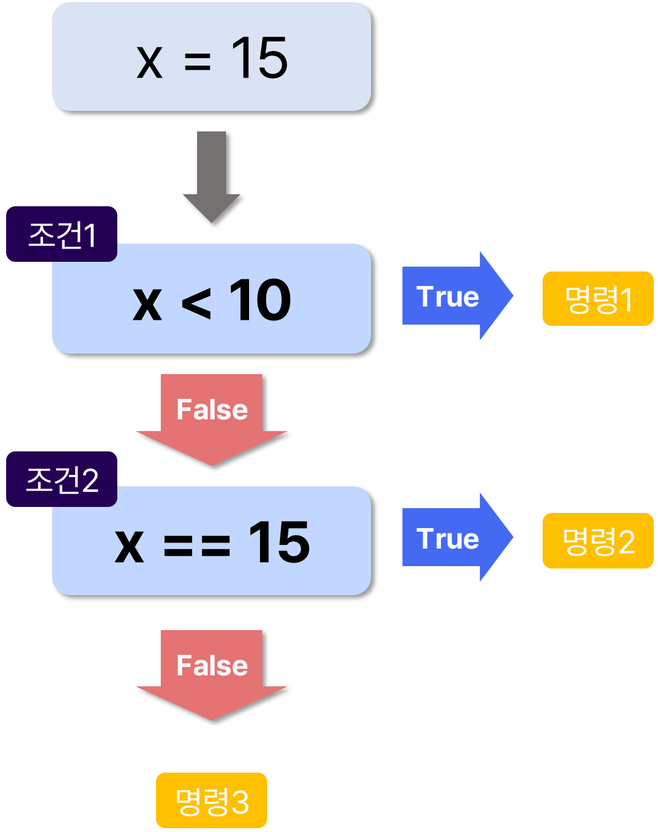
```
if 조건 1:
....<수행할 명령>
elif 조건 2:
....<수행할 명령>
```

```
x = 15
if x < 10:
....print("x는 10보다 작습니다.")
elif x == 15:
....print("x는 15입니다.")
else:
....print("x는 10보다 크고 15와 다릅니다.")
```

if - elif문

- 조건 1이 **True**면 if문
- 조건 1이 **False**이면서 조건 2가 **True**면 elif문 실행

```
x = 15
if x < 10:          조건1
....print("x는 10보다 작습니다.")      명령1
elif x == 15:       조건2
....print("x는 15입니다.")            명령2
else:
                                      명령3
....print("x는 10보다 크고 15와 다릅니다.")
```



if - elif문

```
x = 15
if x < 10:
....print("x는 10보다 작습니다.")
elif x == 15:
....print("x는 15입니다.")
else:
....print("x는 10보다 크고 15와 다릅니다.")
```

정리 : if-elif-else 문

```
if 조건 1:
    do A
elif 조건 2:
  do B
elif 조건 3:
  do C
. . .
else:
  do X
```

조건 1 **True**

→ A 실행

조건1 **False and** 조건2 **True**

→ B 실행

조건1 **False and** 조건2 **False and** 조건3 **True**

→ C 실행

...

모든 조건이 **False**

→ X 실행

for 반복문

1, 2, 3, 4, 5 페이지에서

시퀀스

제품명을 저장해서

for

출력!

명령

for 반복문

- **원소로 반복**하는 방법
- 시퀀스의 원소를 **하나씩 변수**에 넣어가면서 명령 실행

```
for 변수 in 시퀀스:
....<수행할 명령>
```

```
fruits = [“사과”, “바나나”, “체리”]

for fruit in fruits:

....print(fruit)
```

for 반복문

- for문에 들어갈 명령들은 **같은 들여쓰기**로 구분!

```
for 변수 in 시퀀스:
....<수행할 명령>
```

```
fruits = [“사과”,”바나나”,”체리”]
for fruit in fruits :
....print(fruit)
```

for 반복문 예시

- 1, 2, ..., 10까지 출력하기

```
for i in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
....print(i)
```

for-range 반복문

1~10페이지에서

범위

제품명을 저장해서

for

출력!

명령

range란

- **연속되는 숫자**를 만들어 주는 시퀀스 자료형

```
range(a, b) #a, a+1, a+2, … , b-1
range(0, 9) #0, 1, … , 7, 8
range(5) #range(0, 5) - 0, 1, 2, 3, 4
```

for-range 반복문 1

- **구간으로 반복**하는 방법
- **a이상 b미만의 수**를 변수에 넣어가면서 명령을 수행

```
for 변수 in range(a,b):
....<수행할 명령>
```

```
a = [1]
for i in range(2, 4):
....a.append(i)
print(a) #[1, 2, 3]
```

for-range 반복문 2

- **횟수로 반복**하는 방법
- **a번** 만큼 명령을 수행

```
for 변수 in range(a):
....<수행할 명령>
```

```
count = 0
for i in range(10):
....count = count + 1
print(count) #10
```

while 반복문

크롤링 개수가 1000보다 작은

조건

동안

while

데이터를 저장!

명령

while 반복문

- **조건으로 반복**하는 방법
- **조건이 True**이면 명령을 수행

```
while 조건:
....<수행할 명령>
```

```
i = 5
while i>0:
....print(i)
....i = i - 1
print(“Launch!”)
```

while 반복문 예시

```
i = 1
sum = 0
while i<5:
....sum = sum + i
....i = i + 1
print(sum) #10
```

1부터 4까지 더하기

while문에서 조건이 항상 True면?

- 무한정 코드가 실행된다. 도르마무처럼!

```
i = 1
while i>0: #항상 True
....print(i)
....i = i + 1
```

```
# 실행결과
1
2
3
…
```

break문

- **if문**으로 조건을 걸어준 다음, break 실행
- 반복문을 **탈출**하는 역할!

```
i = 0
while True:
    print(“knock”)
    if i >= 3:
        break
    i = i + 1
```

```
# 실행결과
knock
knock
knock
knock
```

리스트(List)

- **여러 자료를 순서대로 보관**하는 자료형.
- 다른 종류의 자료를 **함께** 담을 수 있음
- 각각의 **위치**를 **0**부터 순서대로 매길 수 있습니다(인덱스)

```
[] # 빈 리스트
['a', 'b']
['a', 2] # 다른 자료형을 함께!
```

```
[2, 4, 6, 8]
 0  1  2  3
```

append

- 추가할 자료를 **리스트 마지막 원소 뒤**에 추가
- 오직 한 개의 자료만 넣을 수 있음

```
a = []
b = [“a”, “b”, “c”]

a.append(10)
b.append(“d”)

print(a, b)
```

실행 결과

```
[10] [“a”, “b”, “c”, “d”]
```

insert

- 추가할 자료를 **지정한 인덱스 위치 i**에 추가
- 오직 한 개의 자료만 넣을 수 있음

```
c = [1, 2, 4, 5]
c.insert(2, 3)


print(c)
```

실행 결과

```
[1, 2, 3, 4, 5]
```

remove

- 특정 요소를 제거
- 여러 개가 존재할 경우, **가장 앞**에 있는 요소를 제거함.

```
d = [2, 1, 2, 3, 4]
d.remove(3)


print(d)
```

실행 결과

```
[1, 2, 3, 4]
```

sort

- 리스트 내의 자료들을 **순서대로 정렬**
- 기본은 **오름차순/사전순**

```
num = [7, 2, 5, 3]
fruits = [“coconut”, “apple”, “banana”]

num.sort()
fruits.sort()
print(num, fruits)
```

실행 결과

```
[2, 3, 5, 7][“apple”, “banana”, “coconut”]
```

sort

- sort(**reverse=True**) : **내림차순**

```
num = [7, 2, 5, 3]
fruits = [“coconut”, “apple”,
“banana”]


num.sort(reverse=True)
fruits.sort(reverse=True)
print(num, fruits)
```

실행 결과

```
[7, 5, 3, 2][“coconut”,
“banana”, “apple”]
```

시퀀스(sequence) 자료형

- 데이터를 **순서대로** 나열한 자료형
- 예시 : 리스트, 문자열, 등등
- 리스트의 상위 개념

1. 인덱싱

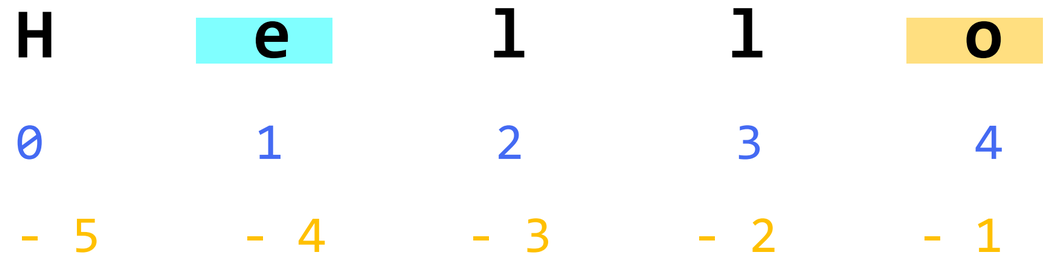
index를 이용해서 리스트나 문자열의 **특정 위치의 원소**를 가져오는 방법

- **string/list[index_number]**

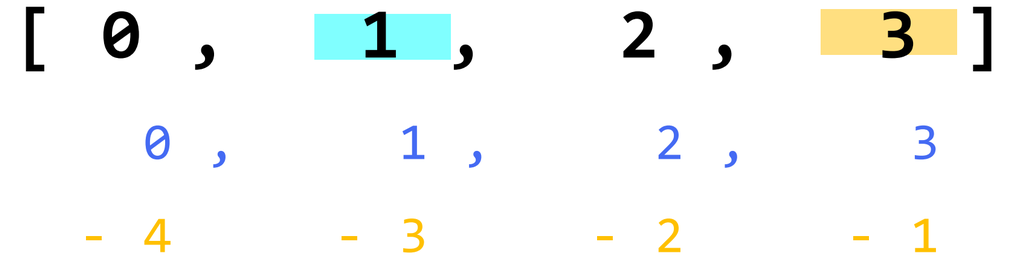
```
# alpha에서 인덱스 1인 원소 'e'를 출력
alpha = "Ready"
print(alpha[1])
>>> e
```

1. 인덱싱

- 뒤에서 인덱스 순서는 음수로 설정함.




```
words = “Hello”
numbers = [0, 1, 2, 3]


print(words[1], numbers[1])
# 뒤에서 1번째 원소
print(words[-1], numbers[-1])
```



실행 결과

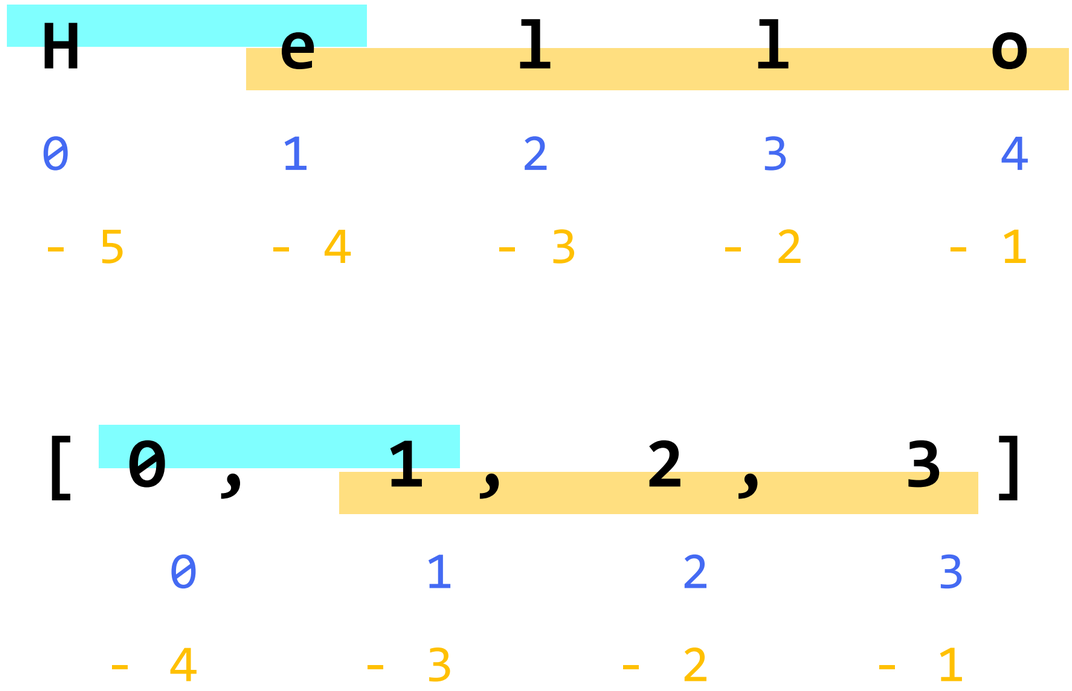
```
e 1
o 3
```

2. 슬라이싱

- index를 이용해서 리스트나 문자열의 일부분을 잘라서 가져오는 방법
- **string/list[a**(시작 인덱스)**:b**(종료 인덱스)**]** : **a**번째 원소 **이상 b**번째 원소 **미만**

```
# beta에서 2번째 원소 이상, 5번째 원소 미만을 가져온다.
beta = [2, 4, 6, 8, 10, 12, 14]
print(beta[2:5])
>>> [6, 8, 10]
```

2. 슬라이싱

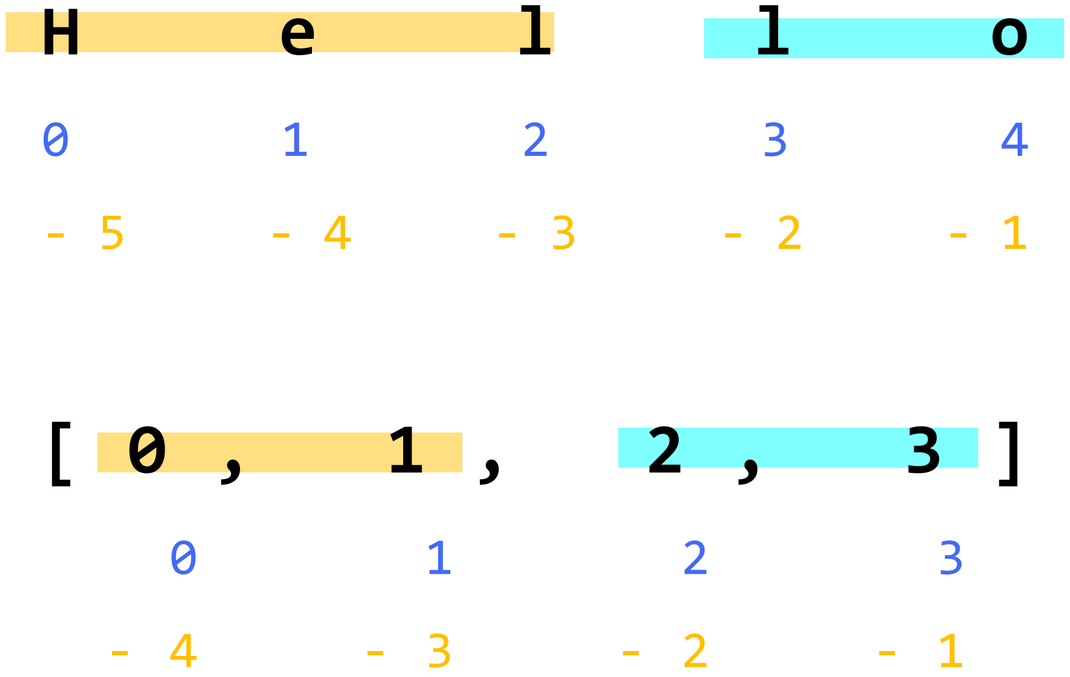



```
words = “Hello”
numbers = [0, 1, 2, 3]


# 처음~2번째 슬라이싱
print(words[:2], numbers[:2])
# 2번째~끝까지 슬라이싱
print(words[1:], numbers[1:])
```

실행 결과

```
He [0, 1]
ello [1, 2, 3]
```

2. 슬라이싱




```
words = “Hello”
numbers = [0, 1, 2, 3]

# 뒤에서 2번째~끝까지 슬라이싱
print(words[-2:], numbers[-2:])
# 뒤에서 2번째 전까지 슬라이싱
print(words[:-2], numbers[:-2])
```

실행 결과

```
lo [2, 3]
Hel [0, 1]
```

3. in 연산자

- 시퀀스 안에 원소가 있는지 확인

```
words = “Hello”
numbers = [0, 1, 2, 3]

print(“o” in words)
print(4 in numbers)
```

실행 결과

```
True
False
```

4. len 연산자

- 시퀀스 자료형의 길이 확인 가능

```
words = “Hello World”
numbers = [0, 1, 2, 3]

print(len(words))
print(len(numbers))
```

실행 결과

```
11
4
```

5. +/* 연산자

- 시퀀스 자료를 이어 붙이거나 반복 가능

```
front = “py”
back = “thon”
first = [0, 1]
second = [2, 3]


print(front + back)
print(first + second)
```

실행 결과

```
python
[0, 1, 2, 3]
```

5. +/* 연산자

- 시퀀스 자료를 이어 붙이거나 반복 가능

```
words = “go!”
num = [2, 3]

print(words * 3)
print(num * 3)
```

실행 결과

```
go!go!go!
[2, 3, 2, 3, 2, 3]
```

튜플 (Tuple)

- **소괄호()**를 사용해 여러 값을 순서대로 저장할 수 있는 시퀀스 자료형
- 인덱스를 통한 접근 가능
- **변경/삭제 불가능**

```
my_tuple = (1, 2, 4, “python”)


print(my_tuple[1])
print(my_tuple[-1])
print(my_tuple[2:])
```

실행 결과

```
2
python
(4, ‘python’)
```

튜플 (Tuple)

- 변경/삭제 불가능

```
my_list = [1, 2, 4]
my_tuple = (1, 2, 4, “python”)


my_list[2] = 3
print(my_list)
my_tuple[2] = 3 # ❌ TypeError 발생
```

실행 결과

```
[1, 2, 3]
```

튜플 (Tuple)

```
empty_tuple = ()  # 빈 튜플


t = (5) # 그냥 정수 5
t = (5,) # 1개짜리 튜플
```

세트 (Set)

- **중복 없는** 자료의 집합
- **순서가 없는** 자료형 (인덱싱 불가)
- 중괄호 **{}**
- 집합 연산

```
my_set = {1, 1, 3, “python”}

print(my_set)
```

실행 결과

```
{1, 3, ‘python’}
```

세트 (Set)

- set() 함수로 생성
- 인덱싱 불가함으로 반복문으로 접근 가능함

```
s1 = set() # 빈 세트 생성


my_set = {1, 2, 3}
for element in my_set:
....print(element)


print(2 in my_set)
```

```
실행 결과
1
2
3


True
```

세트 (Set)

- 원소 추가 : **add**(), **update**()
- 원소 삭제 : **remove**()

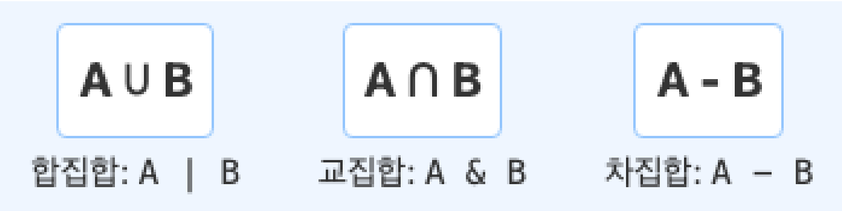
```
my_set = {1, 2, 3}
my_set.add(4) # 단일 요소 추가
print(my_set)
my_set.update([5, 6]) # 여러 요소 추가
print(my_set)
my_set.remove(3)
print(my_set)
```

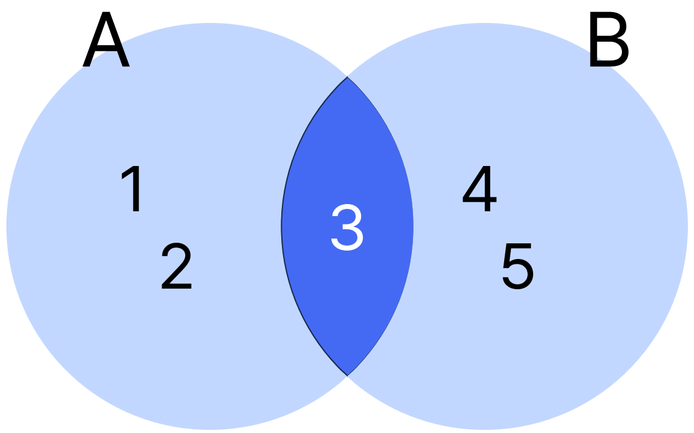
실행 결과

```
{1, 2, 3, 4}
{1, 2, 3, 4, 5, 6}
{1, 2, 4, 5, 6}
```

세트 (Set)

- 세트 연산 : **합집합**(A | B), **교집합**(A & B), **차집합**(A - B)



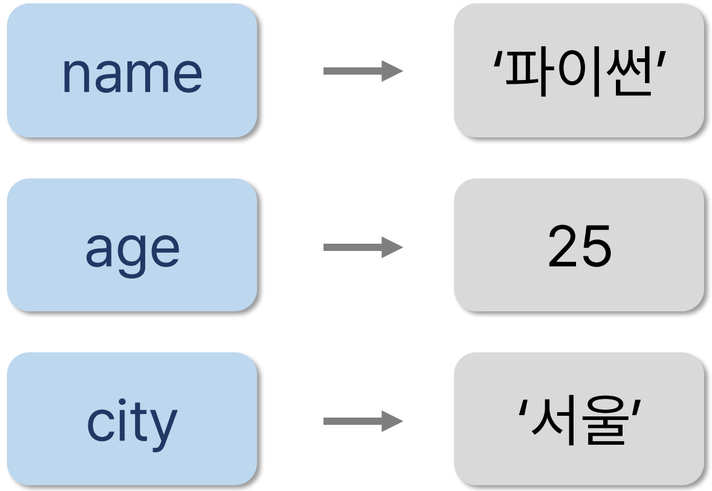



```
A = {1, 2, 3}
B = {3, 4, 5}
print(A | B) # {1, 2, 3, 4, 5}
print(A & B) # {3}
print(A - B) # {1, 2}
```

딕셔너리 (Dictionary)

- **키(key)와 값(value) 쌍**으로 데이터를 저장하는 자료형
- 중괄호{}와 콜론(:)을 사용하여 키-값 쌍을 표현
- 중복된 키 허용하지 않음 = **고유한 키**
- **중복 값 가능**
- 순서 보장(python3.7 이후 지원)
- 값 수정 가능

```
my_dict = {'name': '파이썬',
'age': 25, 'city': '서울'}
```



딕셔너리 (Dictionary)

- **dict()** 또는 **중괄호{}**로 생성
- 키를 통해 접근할 시 키가 없으면 에러 발생함.

  ➡️ **get(키, 기본값)** 활용

```
# 딕셔너리 생성
user = {'name': '김철수', 'age': 25}
empty_dict = {}
users= dict(name='홍길동', age=30)

# 키를 통한 접근
print(user['name'])  # 김철수
print(user.get('c', '기본값') # 기본값
```

딕셔너리 (Dictionary)

- **update()** : 여러 항목을 추가 또는 수정 가능

```
# 새 키-값 쌍 추가
user['phone'] = '010-1234-5678’
# {‘name’: ‘김철수’, ‘age’: 25, ‘phone’:
‘010-1234-5678’}

user.update({‘name’: ‘이철수’,
‘gender’: ‘M’}) # 여러 항목 추가/수정
```

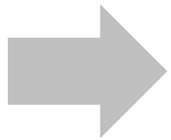
- **del / pop() / clear()** : 삭제

```
del user[‘phone'] # 특정 키-값 쌍 삭제

# 삭제 후 값 반환
value = user.pop(‘gender')
# value = ‘M’

d.clear() # 모든 항목 삭제 (초기화)
```

리스트 컴프리헨션(List Comprehension)

- 리스트를 간결하고 효율적으로 생성하는 문법
- for문을 한 줄로 압축
- 코드 가독성과 생산성 향상

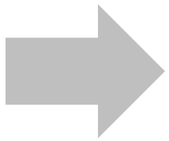
```
for 변수 in 반복가능한_객체:
....표현식
```



```
[표현식 for 변수 in 반복가능한_객체]
```

리스트 컴프리헨션(List Comprehension)

```
numbers = []
for i in range(1, 6):
    numbers.append(i)
```



```
numbers = [i for i in range(1,
    ]
```

실행 결과

```
[1, 2, 3, 4, 5]
```

리스트 컴프리헨션(List Comprehension)

```
numbers = []
for i in range(1, 6):
    if i % 2 == 0:
        numbers.append(i)
```

실행 결과

```
[2, 4]
```

```
numbers = [i for i in range(1, 6) if i % 2 == 0]
```

딕셔너리 컴프리헨션(Dictionary Comprehension)

```
fruits = ["apple", "banana", "cherry"]
fruit_lengths = {}
# 과일 이름을 키로, 길이를 값으로
for fruit in fruits:
    fruit_lengths[fruit] = len(fruit)
```

실행 결과
```
{'apple': 5,
'banana': 6,
'cherry': 6}
```

```
fruit_lengths_comp = {fruit: len(fruit) for fruit in fruits}
```

파이썬 객체와 메모리

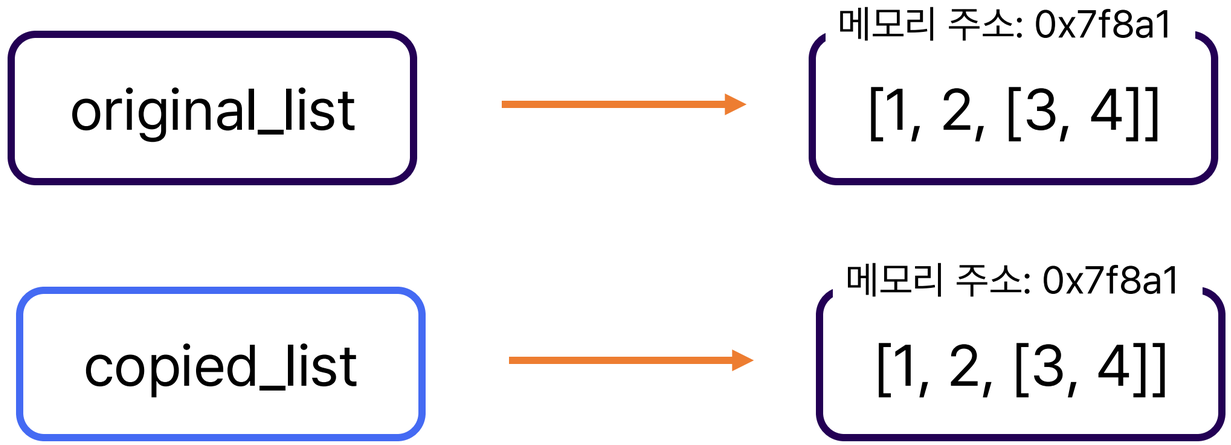
- 파이썬의 객체는 메모리 상에서 **참조**(reference)로 관리됨
- 변수는 실제 데이터가 아닌 **데이터의 주소**를 가리킴

my_list = [1, 2, 3, 4]

그냥 대입하기 (=) 의 문제점

- 리스트의 **참조(reference)**만 복사 ➡️ 두 변수가 **같은 리스트 객체를 가리킴!**

```
original_list = [1, 2, [3, 4]]
copied_list = original_list
```

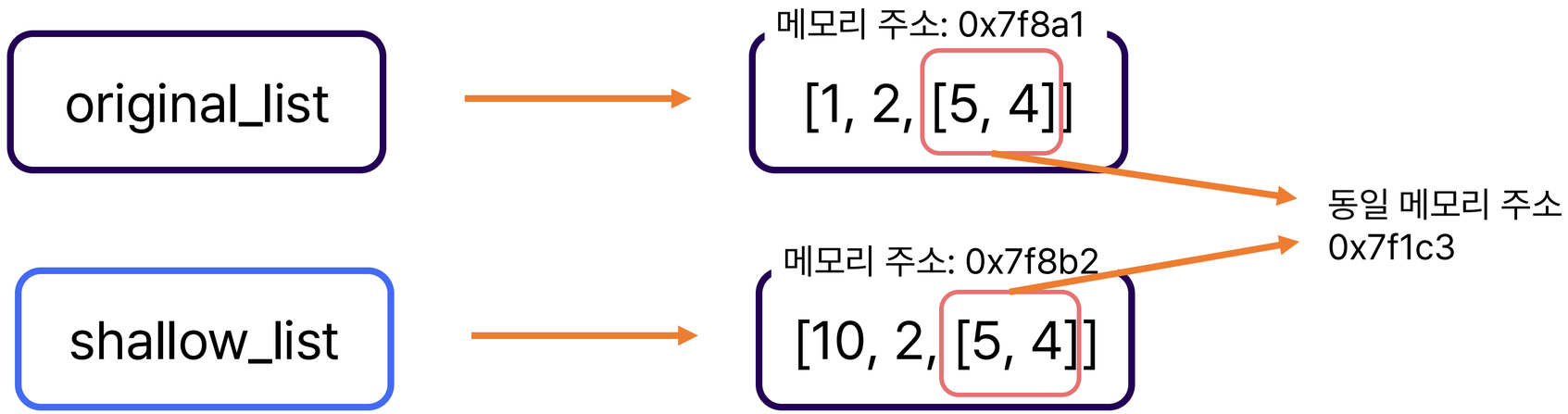


얕은 복사(Shallow Copy)

- 원본 객체의 **내용만** 복사하고, 중첩된 객체는 **참조만** 복사

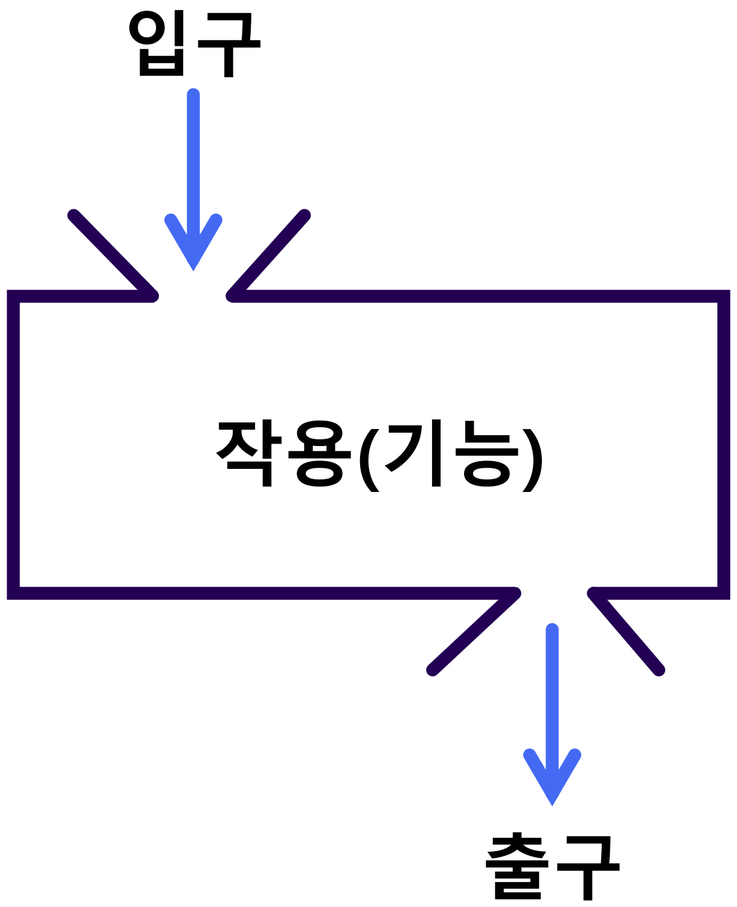
```
original_list = [1, 2, [3, 4]]
shallow_list1 = original_list[:] # 방법 1
shallow_list2 = list(original_list) # 방법 2
```

```
import copy
shallow_list3 = copy.copy(original_list) # 방법 3
```



함수의 배경지식 : 프로그래밍의 기본 틀 구조

- 컴퓨터에게 정보를 입력하고, 컴퓨터는 작업 결과를 출력



함수

- **특정 기능**을 수행하는 코드의 모임

len()	자료를 넣으면 그 **자료의 길이**를 알려준다
int()	자료를 넣으면 **정수형으로 변환**해서 알려준다
str()	자료를 넣으면 **문자열로 변환**해서 알려준다

내장 함수

- 파이썬 개발자들이 **이미 만들어 둔 함수**들
- 편리하게 가져다 쓰면 된다

len()	자료를 넣으면 그 **자료의 길이**를 알려준다
int()	자료를 넣으면 **정수형으로 변환**해서 알려준다
str()	자료를 넣으면 **문자열로 변환**해서 알려준다

함수 만들기

- **def**ine(정의하다) 키워드를 이용해서 함수 정의
- 같은 들여쓰기로 명령 작성
- **return**을 이용해서 함수 외부로 값을 전달

```
def 함수이름(매개변수):
____<수행할 명령>
____. . .
____return 반환값
```

Method(매서드)

- **특정 자료**에 대해 **특정 기능**을 하는 코드
- 함수는 **특정 기능**을 한다 (매개변수를 이용해 자료를 전달)
- 매서드는 **특정 자료와 연관 지어** 기능을 한다(**.을 찍어** 사용)

```
my_list = [1, 2, 3]
my_list.append(4)
my_list.count(2)
my_list.pop()
```

```
my_list = [1, 2, 3]
len(my_list)
sum(my_list)
min(my_list)
```

모듈과 패키지의 필요성

모듈 → 특정 목적을 가진 함수, 자료의 모임

"누군가 만들어놓은 함수, 변수 등을 활용하자!"

패키지 → 모듈을 편리하게 관리하기 위해서

모듈 불러오기

import (불러오다) 키워드로 모듈 사용

```
import random

# random 모듈 불러오기
```

모듈 활용

불러온 모듈 속 함수, 변수를 활용

```
# cal.py
def plus(a, b):
    c = a + b
    return c
```

```
# main.py
import cal
print(cal.plus(3,4))
# 7
```

모듈 활용

불러온 모듈 속 함수, 변수를 활용

```python
# cal.py
def plus(a, b):
    c = a + b
    return c
```

```python
# main.py
import cal
print(cal.plus(3,4))
# 7
```

```python
# main.py
from cal import plus
print(plus(3,4))
# 7
```